Introduction

This capstone project is a study of the principle of self-interpretation, where the aim is to design and implement a self-interpreter for the Scheme programming language, i.e. a Scheme interpreter written in Scheme. Scheme was chosen to be the programming language that the self-interpreter will be written in due to its expressive power and minimalism, which will allow us to better explore the principles of self-interpretation. The purpose behind the implementation of such a self-interpreter is to explore the richness of the expressivity of the programming language used, and to also study the computational aspects of such self-interpreters, e.g. its efficiency.

Once the implementation of the self-interpreter in Scheme is complete, we will be able to measure the computational cost of layering software (e.g. a processor running a virtual machine running a simulator running yet another virtual machine and so on...) in the idealized setting where all the levels of the software are identical (Scheme and its self-interpreters, or a tower of Scheme self-interpreters). Furthermore, we can also measure the additional costs incurred due to Scheme’s non-static typing such as the type-checking of each argument done in the implementation of the self-interpreter, via implementing a self-interpreter that assumes that all its input is of the correct type and comparing this with one that does the necessary type checks.

An interpreter for Scheme written in OCaml

Before starting on the implementation of the Scheme self-interpreter, I first started on the implementation of an interpreter for Scheme written in OCaml first. This additional step is done for an easier implementation of the self-interpreter for Scheme, where instead of implementing the Scheme self-interpreter *from scratch*, we will have a reference (and working) interpreter for Scheme written in OCaml, which can then be used for transliteration to implement the Scheme self-interpreter.

OCaml was chosen as the metalanguage of the initial interpreter for Scheme, due to my familiarity of with the language as well as the following *important* characteristics of OCaml:

1. Static-typing: OCaml’s nature of being statically-typed is greatly useful in the implementation of an interpreter that is *guaranteed* to be free from **type-errors**, which can be a real nuisance and creep in as run-time errors if the metalanguage for the interpreter was a language that is dynamically typed instead.
2. Pattern Matching: OCaml’s ability to do *pattern matching* on otherwise complicated data types can be said to be a blessing for interpreter/compiler writers. The ability to completely de-structure a data type into its constituent constructors allows us to have a high level of *control* over the manipulation of data types, where we get to perform different actions for different constructors for a *single data type.* This is particularly valuable in the implementation of an interpreter, where we have to de-structure each expression of the object language (in this case Scheme) and implement it in the meta language (in this case OCaml) according to its grammar. OCaml’s ability to further detect non-exhaustive / erratic pattern matchings adds icing onto the cake.
3. Easy recursion over tree-structured datatypes: This is a consequence of OCaml’s ability to do pattern matching over datatypes, and allows for the writing of recursive functions over datatypes to be simple and straightforward.

The implementation of an interpreter for Scheme in OCaml will require the following ingredients:

1. Implementation of the Grammar of Scheme in OCaml via constructing our own types,
2. Implementation of the expressible values of the Scheme language, or the possible types that evaluated Scheme programs can take,
3. Implementation of the *environment*, which stores the definition of variables and allows for the user to re-use the values of these stored variables via the invocation of these variables’ names,
4. Implementation of *primitive* functions, which are functions that exist as native built-in functions inside the environment that can be used by the user,
5. Implementation of the interpreter, which recursively evaluates *parsed* Scheme programs into its result,
6. Implementation of *unit tests*, which rigorously tests for each of the above parts to ensure that the interpreter is working as expected.

The following sections then go on to explain the implementation of each of the above 6 parts.

Grammar of Scheme and Expressions

The following figure illustrates the ***simplified*** grammar of Scheme, written in *BNF*:

Table

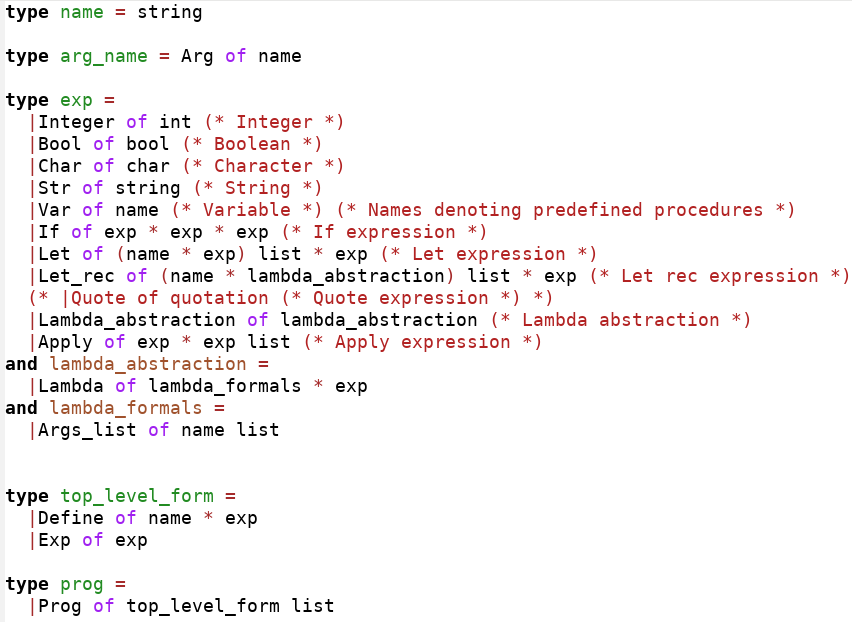
Description automatically generated

Figure 1: Modified figure of the operationalized BNF of Scheme from <https://users-cs.au.dk/danvy/dProgSprog16/Lecture-notes/week-4_a-syntax-checker-for-Scheme.html#pure-scheme-spelled-out:let-expression>, used and modified with permission from Olivier Danvy, the owner of the website.

where the use of Kleene stars and pluses have been operationalized.

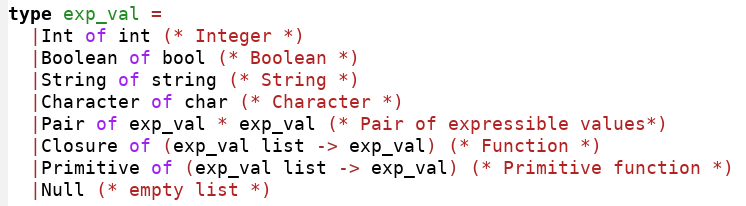
The above BNF of Scheme removes some often-seen constructs from Scheme like the and-expression, or-expression, cond-expression, case-expression etc, because the aforementioned constructs are simply syntactic sugar provided by Scheme that can be expressed via the use of the if-expression in Scheme. This removal of syntactic sugar constructs allows for the implementation of an interpreter for a smaller yet still complete version of Scheme, which is essential given that we only want to focus on the core features of Scheme for the implementation of the interpreter.

The implementation of the operationalized and simplified BNF in OCaml is then as follows, where the quotation type has been left out for now:



Expressible Values

With the above BNF of Scheme defined in OCaml, we can then proceed to define the various types of *expressible values*, or the possible types of results we can get via interpreting Scheme programs, as follows:



Note that in the above definition of expressible values, there exist two types of functions, namely Closures and Primitives. Closures refer to any generic function defined by the user in Scheme, whilst Primitives refer to pre-defined functions that exist in the *environment* of Scheme that can be used by the user without redefinitions. Such pre-defined functions include common operators over integers, Booleans, pairs and the other types of expressible values.

We note that the definition of our String and Pair expressible values are immutable in this case, which prevents the use of set! Functions that mutate the values of the string or pair, unlike the original String and Pair values in Scheme which are mutable via the set! operation defined in Scheme. This choice of immutability for our Pair expressible values is for the prevention of circularly defined lists, which will require additional cumbersome checks in the definition of list-length functions; whilst the choice of immutability for our String expressible values is for consistency (i.e., the use of immutable values for all expressible values) and due to the lack of necessity in implementing mutable Strings for the interpreter to work.

We further note that because Lists in Scheme are constructed via the use of Pairs, we have also left out the List type of expressible value since it is expressed via Pairs in Scheme. A *list* in Scheme is defined to be either the empty list, or the defined Null value; or a Pair whose cdr is a list. A ***proper*** *list* in Scheme is a list that has the empty list as its final element, whilst an ***improper*** *list* in Scheme is instead a list where the final element is **not** the empty list. This distinction between proper and improper lists will matter later on when we implement our Primitive functions for lists, where the list-length and list-ref functions will check for whether the input list is proper or improper before returning a result or an error message.

The remaining types of expressible values, like integers, strings, characters, booleans and the empty list denoted by Null are straightforward in their implementation.

Environment (and its related functions)

As seen from the above defined grammar of Scheme, Scheme supports the use of *variables*, which are essentially named denotations of evaluated Scheme expressions or expressible values. For our interpreter to be able to keep track of names and the expressible values they denote, we require the use of an *environment*, which is some sort of memory that keeps track of these name-value pairs.

In our implementation of the interpreter, we choose to use a simple association list (or a list of pairs), where the first element of the pair stores the name of the variable with the second element of the pair storing the expressible value denoted by the name. The type of our environment is then simply:



Our empty environment can then simply be defined as the empty list:



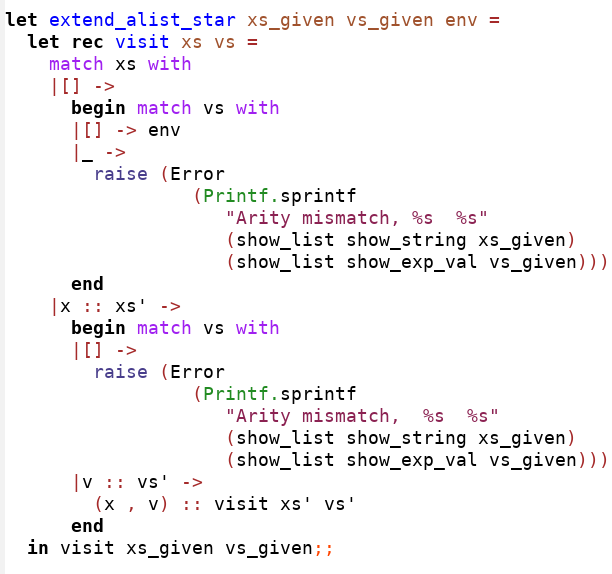
With the environment defined, we can then define functions that serve to:

1. *Extend* the environment, or adding name-value pairs into the environment,
2. *Lookup* for the denotation of a name if it exists inside the environment

The basic function that extends the environment simply takes in the name of the variable and its denotation, and returns an environment that has the pair containing the input name and denotation cons-ed onto the original input environment:



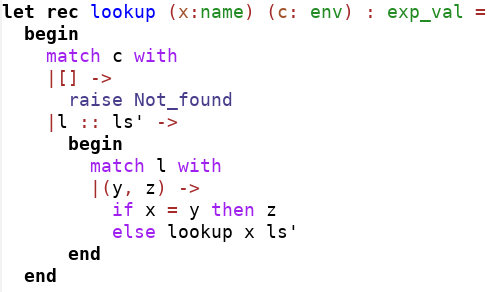
Beyond this simple implementation, we can define a function that takes in a *list* of names and a *list* of denotations instead which adds each name and denotation from the 2 lists elementwise and cons them into the input environment:



The above function is recursive in nature and recurses over the list of names, with the inductive specification being as follows:

1. Base Case: If the input list of names is the empty list, and if the input list of denotations is also the empty list, we simply return the environment. Otherwise, if the input list of denotations is not the empty list, we raise an error due to the differing lengths of the 2 input lists.
2. Induction Step: If the input list of names can be represented as x :: xs’, where x is the head of the list and xs’ is the tail of the list, if the input list of denotations is the empty list, we raise an error due to the differing lengths of the 2 input lists. Otherwise, if the input list of denotations can be represented as v :: vs’, where v is the head of the list of denotations and vs’ is the tail of the list of denotations, the result is simply (x, v) cons onto the induction hypothesis, which is the result of the function applied on xs’ and vs’.

For the lookup function, the implementation is simple where we simply recurse over the environment and find the first instance of the pair containing the name and return its corresponding denotation from said pair; otherwise if we have recursed through the entire environment and don’t find any matching pairs, we raise the Not\_found error to indicate that the name does not exist in the environment:



(Maybe can talk about possible improvements where we implement the environment as a function instead, and ensure no repetition or smth?)

Primitive functions

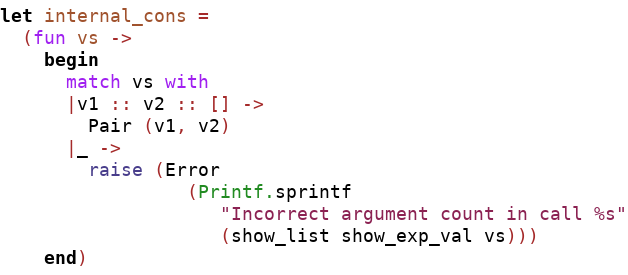
With the environment and its functions defined, we can go ahead to define the primitive functions (or in-built functions) of Scheme and initialize these functions inside our initial environment for use. The in-built functions for Scheme can be grouped based on:

1. The type of *object* the function acts on. Example objects include integers, strings, characters, Booleans, pairs and lists.
2. The *arity* of the function, i.e., the number of valid arguments the function can take. Some functions have no defined arity and can take in any number of arguments, whilst some functions have fixed arity and are only defined for, as an example, one or more arguments. We further note that in Scheme, procedures (or functions) are *uncurried*, and their arguments are naturally stored in a list. This essentially means that the number of arguments supplied to the function is equal to the number of elements inside the list. Hence, testing whether the number of arguments supplied to the function is correct involves using OCaml’s pattern matching to check for the number of elements in the list storing the arguments.
3. The type of the function, i.e. whether it is a comparison function, an indexing function, a function that creates an object from multiple other objects etc.

Given the large number of essential primitive functions to define, this section will only elaborate on a representative sample of the primitive functions that we have defined for the interpreter.

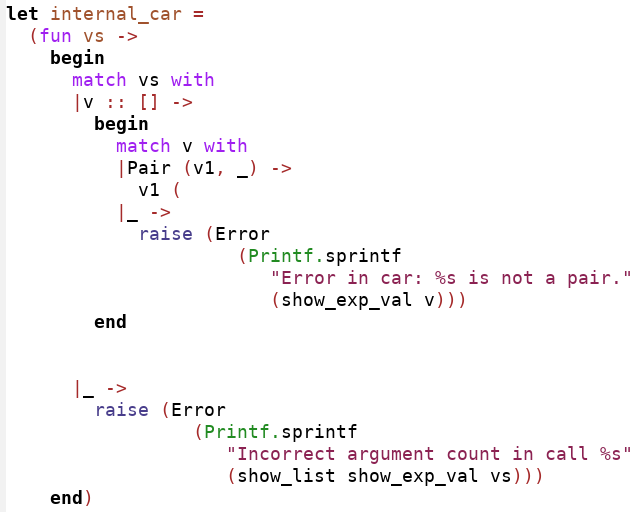
Functions that act on Pairs

An example function that acts on pairs would be the binary function Cons, which takes in two arguments and returns a Pair with the first input argument as its car and the second input argument as its cdr:



where a suitable error message printing the arguments supplied (via the use of an unparser) if the number of arguments is incorrect is shown.

Another function that is similar but involves type-checking is the car function that returns the first element contained inside the pair:



The car function is unary and is only defined for pairs. Hence, as per its definition in Scheme, the function first checks for whether the number of arguments supplied to it is correct (and printing an error message if the number of arguments is incorrect), before checking for the type of argument. If the type of argument is a pair, the function returns the car of the argument; otherwise the function will return an error stating that the supplied argument is of an incorrect type, i.e., not a pair, using our pre-defined unparser.

The remaining functions for pairs are similar to the above two with the only exception being their functionality.

Functions that act on Integers

For integers, most of the functions that act on them are functions that do mathematical operations or comparisons. Hence, although we would expect these functions to only take in 2 arguments, most of them are however defined to have no fixed arity (variadic) with only a few of them accepting just 1 or 2 arguments.

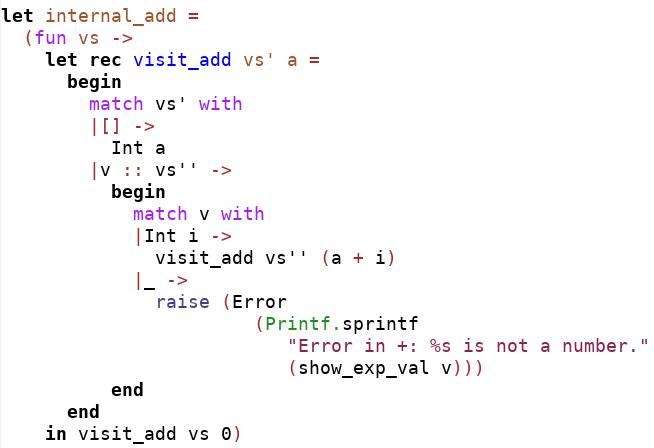
For this section, we would provide 3 representative functions, namely 1) A function that does mathematical operations which is variadic in nature, 2) A function that also does mathematical operations which is binary /dyadic in nature, and 3) A unary function that checks for the nature of the integer argument.

Addition Function

An example of a function that does mathematical operations and is variadic in nature would be the addition function. In Scheme, the addition function is variadic and simply returns the result of adding all its arguments if they are all integers or returning an error if any one of them is not an integer. Specifically:

1. For the case where the addition function is supplied with 0 arguments, the addition function simply returns 0, or the *neutral element* of addition, because adding anything with 0 leaves the result unaltered.
2. If the addition function is supplied with 1 argument (of integer type), the addition function then returns that argument because it is the result of adding 0 and that argument.
3. If the addition function is supplied with 2 or more arguments (all of integer type), the addition function then returns the result of adding all these arguments together.

We can define such a variadic function in OCaml using a *tail-recursive* function that takes in a list of integers as an argument:



The inductive specification of the above addition function is as follows:

1. Base Case: When the list of arguments is the empty list (i.e., no arguments are supplied), we simply return the accumulator *a* which is initialized to be 0 as per the behavior of addition in Scheme
2. Induction Step: When the list of arguments can be represented as v :: vs’, we first check whether the element v is of type integer. If v is of type integer and can be represented as Integer i, we know that the result of the function is simply our induction hypothesis visit\_add vs’ initialized with the accumulator as *(a + i).*

Quotient Function

An example of a function that does mathematical operations and is binary / dyadic in nature would be the quotient function.

The quotient function simply takes in a list containing 2 integer arguments, and returns the quotient that is a result of dividing the first argument by the second argument:

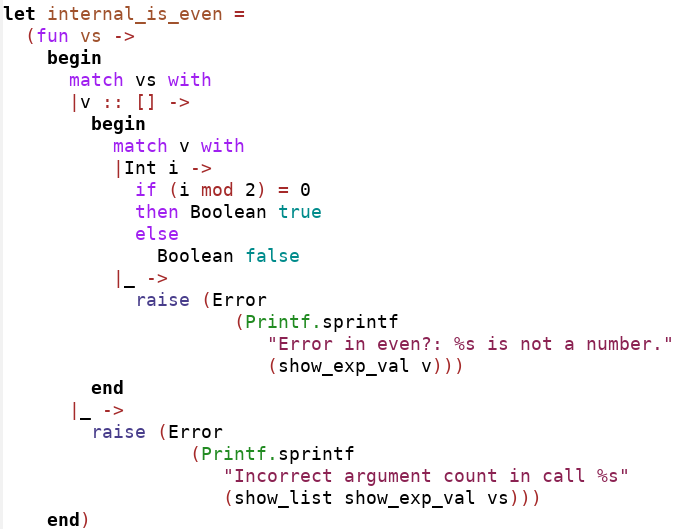


From the above function, we note that the function does a lot of type checking for each of the 2 arguments and returns an error message if either of the 2 arguments is not an integer. We further note the checking of the 2nd integer argument to see whether it is 0, because the division by 0 error will have to be raised for such a case (which in Scheme will raise the same error message we wrote in the function)

Is-even? Function

Finally, an example of a unary function that checks for the nature of the integer argument would be the is-even function.

The implementation of the is\_even function is simple: If there is only 1 argument supplied and it is of type integer, we simply check whether the remainder of dividing said argument by 2 is 0 and return the expressible value Boolean true if it is so and Boolean false otherwise. A wrong number of arguments of a non-integer argument will raise an error:



A large bulk of the remaining functions not illustrated here are variadic functions that test whether the integer arguments provided are monotonically increasing, decreasing or are all equal to each other, and are implemented recursively as well via the use of an accumulator. An example of a comparison function would be provided in the following section for functions operating on Characters, with the implementation being the same modulo the type of object being operated on.

The remaining binary mathematical operators are implemented similarly to addition, albeit with subtraction and division not allowing for the case of 0 arguments (since there is no neutral element for these operations).

Functions that act on Characters

For characters, most of the functions that act on them are functions that do comparisons between them or check that a character belongs to a certain category.

For this section, we would provide 2 representative functions, namely 1) A function that does comparisons between characters which is variadic in nature and 2) A unary function that checks for the type of an input character.

Lesser than Comparison function

The lesser than comparison function checks whether each argument (of type character) is less than the arguments that come after it. In other words, the function checks that the arguments it is supplied with are monotonically increasing and requires at least one argument to be supplied.

Such a variadic function can be defined using tail-recursion over the list, with the inductive specification being as follows:

1. Base Case: In the base case where the list of arguments is empty, we return the accumulator a, which is initialized to be the Boolean value true.
2. Induction Step: If the list of arguments can be represented as v :: vs’, if v is a character c and is greater than the previous character c’ (stored as another argument of the function), the result of the function would be the same as the induction hypothesis (recursing over the remaining list) with the accumulator being the same, and with c being stored as c’ in the function to be compared with the next argument. Otherwise, if c is not greater than c’, we change the Boolean value stored in the accumulator to false (and do nothing if it is already stored as false) and store c as c’ too, before recursing through the remainder of the list to check whether any incorrect arguments exist in the remainder of the list and raise errors if there are any.

The function is then as follows:

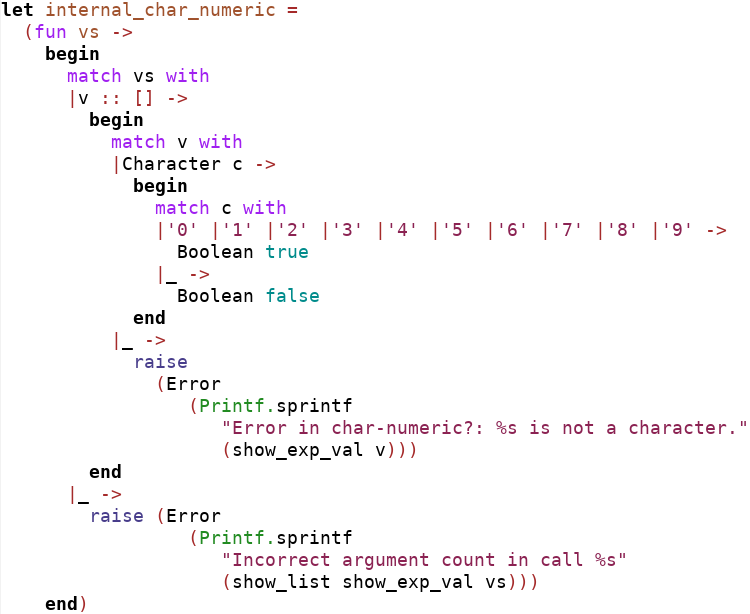


where we raise an error indicating an incorrect argument count if no arguments were supplied, and otherwise initialize the tail-recursive function specified above with the Boolean true as its accumulator and the first character argument of the list as c’. If any of the input arguments are not of type character, an error message indicating type error is raised. All the comparison functions for the integer and string object types work in the same fashion as well.

Is-numeric function

An example of a unary function that checks whether an input character argument is of a specific group is the is-numeric function, which checks whether the input character is numeric.

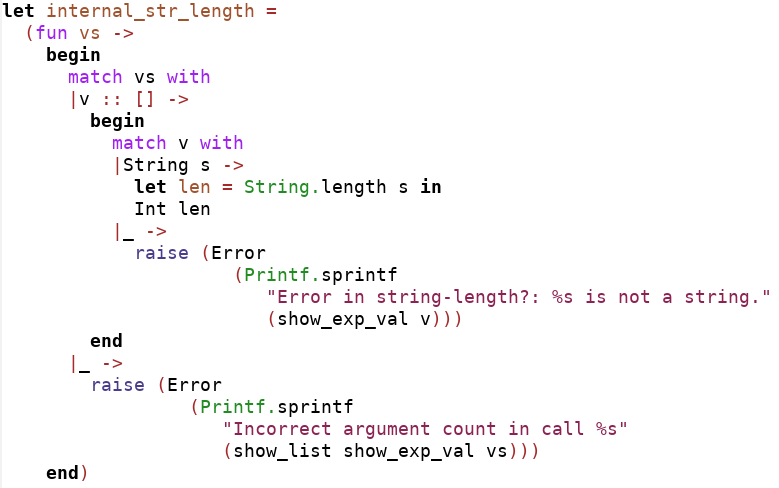
The implementation of such a function is straightforward and simply uses OCaml’s pattern matching:



where as before, errors are raised if the number of arguments is incorrect or if the single input argument is not of type character.

Functions that act on Strings

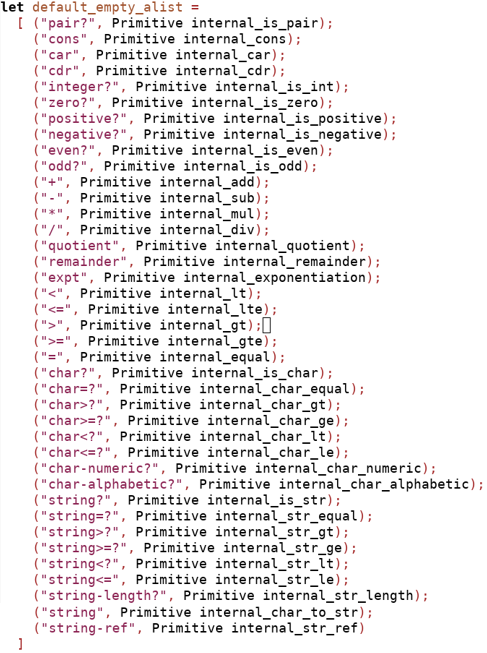
For the case of strings, most of the its inbuilt functions are comparison type functions, with str-ref and str-length being the few exceptions. For a representative sample of functions that act on strings, only the unary function str-length will be illustrated:



As seen in the screenshot above, the implementation of str\_length is very straightforward and is mainly implemented by OCaml’s inbuilt String.length function. As is usual for primitive functions, the number of arguments and type checking occurs as well, with the appropriate errors being raised if required.

As of now, the in-built functions for the list object has been left out in our initial implementation of the Scheme interpreter written in OCaml and will be implemented later on when we are writing our self-interpreter for Scheme.

With our primitive functions defined, our initial environment of the interpreter is then:



Which is simply an association list containing the names of the inbuilt function and their denotations, namely the primitive functions defined.

The Interpreter (Eval) function

With the primitive functions and the environment now defined, we can proceed to implementing the main interpreter function.

The interpreter is a function that takes in a Scheme expression (parsed Scheme code) and performs the expected behavior of the expression according to the specified grammar of Scheme, before returning an expressible value or the result of the computing the Scheme expression. Our interpreter written in OCaml is recursive in nature, and recurses over the AST of the Scheme expressions.

The specification for the OCaml interpreter for Scheme is as follows:

1. Base Case:

If the Scheme expression is:

* 1. An integer n:

An expressible value of type integer with value n will be returned.

* 1. A Boolean b:

An expressible value of type Boolean with value b will be returned.

* 1. A character c:

An expressible value of type Character with value c will be returned.

* 1. A string s:

An expressible value of type String with value s will be returned.

* 1. A variable x:

An expressible value which is the result of looking up x in the current environment will be returned.

1. Induction Step:

If the Scheme expression is:

* 1. An if expression if(e1, e2, e3):

If the result of interpreting the Scheme expression e1 is not the Boolean false, the result of interpreting the Scheme expression e2 will be returned. Else, the result of interpreting the Scheme expression e3 will be returned.

* 1. An apply expression apply(e, es):

If the result of interpreting the Scheme expression e is a Closure function or a Primitive function, and the result of interpreting the list of Scheme expressions es is the list of expressible values vs, then the result of the apply expression is simply the application of the Closure/Primitive function on the list of expressible values vs.

* 1. A lambda expression lambda(formals, body):

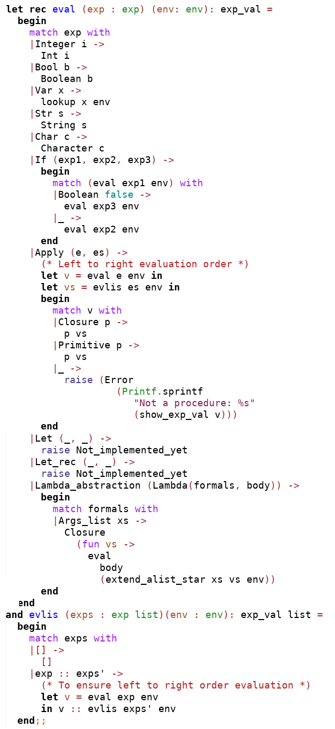
If the formals is simply a list of arguments xs, the result of interpreting the Scheme expression of lambda will be the Closure function that takes in a list of arguments vs, binds them to the list of arguments xs, and evaluates/interprets the body of the lambda expression in the extended environment containing the bindings of the argument names xs to the arguments in list vs.

We note that the let and let-rec constructs have not been implemented for the interpreter yet and will be implemented after the submission of this progress report.

Furthermore, we note that the definition of a helper function will be required for the case where we evaluate the apply expression, or specifically, the list of expressions es in the apply expression. The helper function is simply a function that takes in a list of Scheme expressions and returns a list of expressible values which are the result of interpreting the aforementioned Scheme expressions, and its inductive specification is as follows:

1. Base Case: If the list of Scheme expressions exps is the empty list, then the result is also the empty list.
2. Inductive Step: If the list of Scheme expressions exps can be represented as exp :: exps’, then the result of the helper function applied on exps is the same as cons-ing the interpreted result of exp onto the result of the helper function applied on exps’ (the induction hypothesis).

With the definition of the interpreter and its helper function complete, the implemented interpreter (and helper function) is as follows:



where we set the evaluation order of the interpreter to be from left to right. This left to right evaluation order can be explicitly seen from what happens in the apply clause of the interpreter and the evlis helper function, where we explicitly evaluate the head of the list exps first before evaluating its tail exps’; or when we explicitly evaluate the left argument of the apply expression e before evaluating its right argument es. Not doing any one of the above would lead to an interpreter that has an *inconsistent* order of evaluation, because since OCaml is a programming language that has a right to left order of evaluation, not explicitly evaluating exp first before exps’ in the case of evlis, for example, will lead to the case where exps’ is evaluated first before exp instead in the recursive clause where exps can be represented as exp::exps’.

Testing

With the implementation of the grammar of Scheme, expressible values, the interpreter environment, the primitive functions and the interpreter somewhat completed, we can then proceed to conduct extensive testing for each of these components of our interpreter before we proceed to implement trickier cases such as the let-rec case for our interpreter which has been left intentionally incomplete as of now. This is because once we have an error-free base interpreter, implementing the let-rec case which handles recursive functions will be much easier as the errors raised are now due to errors in the implementation of let-rec and not due to any other buggy feature of the interpreter.

Our testing for the interpreter can then be divided according to the following components:

1. The environment and its related functions
2. The primitive functions
3. The base interpreter as it is

Furthermore, since we want our testing to satisfy code coverage, we not only want to test that our interpreter (and its components) return the correct values when their arguments are error-free, but also want to test that they will return the *correct error* when the arguments are erratic (either in terms of type or in terms of the number of arguments).

The tests can be divided into the following two broad categories:

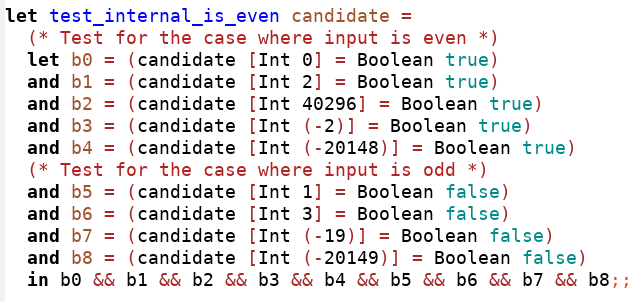
1. Testing that the function gives the correct output given that the input arguments are well-typed and are of the correct number.
2. Testing that the function raises the correct error given either an incorrect type of input argument or an incorrect number of input arguments.

For each of these two categories, we then want to ensure that the testing satisfies code coverage, or that the unit tests do test for each possible branch of our code. In the following paragraphs, we will then provide one simple example for each of the above broad categories.

Testing the is\_even function for correct output

Referring to the is\_even function described in the Primitives section above, we can see that to ensure code coverage for the non-error branches, we will have to test for the case where there is only a single argument which is of type integer, and test for even and non-even numbers.

The following screenshot of the test illustrates the above:

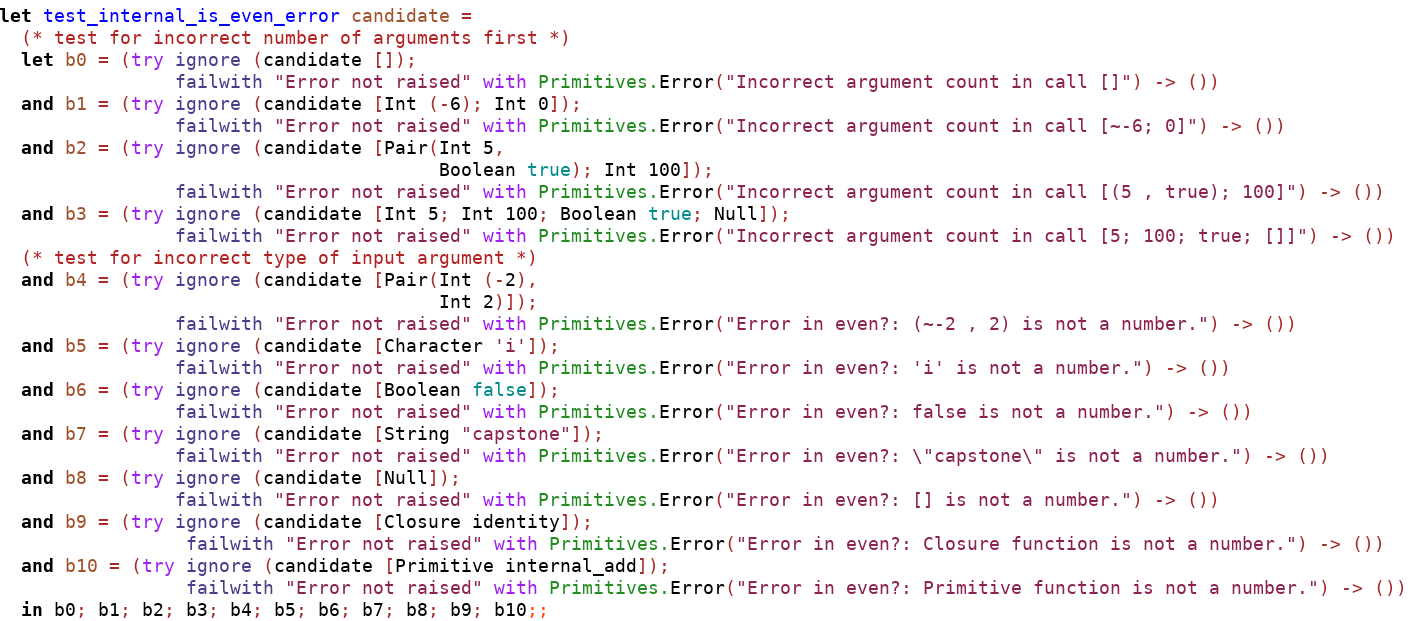


where despite the seemingly simple test function, we have actually satisfied code coverage and also tested for all possible sorts of input integers, namely: 1) Positive numbers, 2) Negative numbers and 3) Zero.

Testing the is\_even function for correct error raising

As mentioned above, we will have to test the is\_even function to ensure that the correct errors are raised. Referring to the internal\_is\_even function above, we know that satisfying code coverage for the function in the case of errors will require our tests to : 1) Test for incorrect arity in the input arguments and 2) Test for incorrect types if the input argument is correct.

The below screenshot then shows the function that tests internal\_is\_even for correct error raising:



Although the test function might seem much longer as compared to the test function for the non-error case, the concept behind both functions are similar. We test for an incorrect arity in the input arguments as seen from test clauses b0 to b3, even inputting the correct input types for the arguments as seen in b1 and ensure that the error “Incorrect argument count in call …” is raised. The same is repeated in clauses b4 to b10, where we test for incorrect input types if there is only 1 input argument (correct arity). Specifically, test clauses b4 to b10 test for each possible type of incorrect expressible value as input to the function, so as to ensure that all inputs of type non-integer will cause a type error to be raised.

Although we only gave two examples earlier, we have applied the above two types of tests extensively for *each* function written inside our implementer to ensure that our implementation of the interpreter is as bug-free as possible. Nevertheless, as Dijkstra mentions, ‘program testing can be used to show the presence of bugs, but never to show their absence!’, we should still be open to the possibility that there *might* still be bugs inside our implementation, and not rest on our laurels even when all our unit tests pass.

A possible area of improvement for testing

From the above examples, the error messages are all printed using OCaml’s Printf.sprintf function which parameterizes the input arguments that cause the error (either via incorrect typing or incorrect arity). However, as seen in the above examples of our test functions, since error messages can also be divided into messages that are a result of incorrect types or incorrect arity, and within these two categories the content of the messages are then the same (modulo the function name calling the error), we can actually further parameterize these errors via declaring Error types. These Error types will then consist of two types, one for incorrect types of arguments and one for incorrect arity, and will take the name of the function as an input, the unparsers required to unparsed our input arguments that cause the error, and the input arguments itself. This will make the writing of functions which raise errors and testing to be much simpler, because we do not then need to type out by hand the expected output of these Errors but rather let the parameterized Error types do the job with the correct arguments.

Further things to do

Having completed most of the Scheme interpreter in OCaml, we are left with the following things to complete for the interpreter:

1. Rewrite the apply expression case and implement call-cc of the interpreter such that it is in continuation passing style (CPS) to mimic the actual application of functions in Scheme which is also in CPS
2. Complete the let expression case of the interpreter, which requires our interpreter to be able to handle locally declared bindings and different scopes
3. Complete the let-rec expression case of the interpreter which will allow for locally recursive functions to be interpreted
4. Extensively test for the above 3 cases as per the previous section, and include tests which utilize our inbuilt Primitive functions

Once our interpreter is complete, we will then transliterate the interpreter written in OCaml to Scheme such that our interpreter for Scheme is now written in Scheme (hence the name self-interpreter), where we will then define the primitive functions for the list object as well as add the quotation type into the grammar of Scheme that we are handling.

Finally, with the self-interpreter in Scheme completed, we will then move our focus towards:

1. Implementing a tower of self-interpreters, i.e., stacking self-interpreters on top of each other to measure the effect that this will have on the runtime of the code. In the process, we will also look to improve the efficiency of our self-interpreter via performing optimizations in the code.
2. Implementing another version of a self-interpreter whose primitive functions do not require any form of type-checking due to the assumption that our inputs are all the correct type. We can then compare the efficiency of this self-interpreter with the efficiency of the original to see the extent of improvement in our runtime if Scheme was a statically typed language as opposed to its current dynamically typed nature. This will then be a good study to see how much faster statically typed languages are as compared to dynamically typed ones.